0100 - Mémoire

Sections, lecture, écriture, alignement, boutisme

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur v252



Rappel

La mémoire (RAM et ROM)

- Connectée au processeur (via le bus)
- Contient des octets (8 bits)
 - Chaque octet a une adresse
 - Chaque octet a exactement une valeur parmi 256 possibilités

Les données

- Sont en mémoire (ou dans quelques registres)
- Représentées sous forme binaire (codage)

Les programmes (langage machine)

- Sont aussi en mémoire
- Représentés sous forme binaire (codage)
- Registre spécial compteur ordinal (pc)
 Contient l'adresse de l'instruction courante

Objectifs

- Comment initialiser les octets en mémoire ?
 - Programmes
 - Et données
- Comment lire et écrire?
 - Instructions machine dédiées
 - Règles et contraintes
- Bonus
 - Afficher et lire des chaînes de caractères

Plan

- 1 Organisation de la mémoire
- 2 Boutisme
- 3 Alignement
- 4 Lectures et écritures
- **5** Tableaux et pointeurs
- 6 Chaînes de caractères
- 7 Conclusion

Organisation de la mémoire

Environnement d'exécution 😵

L'environnement d'exécution détermine

- Comment sont organisés les octets accessible en mémoire
- Comment sont initialisé ces octets
- Valeur initiale du compteur ordinal (et autres registres)

L'environnement d'exécution détermine les règles d'accès

- Quelles sont les adresses valides
 - Car toutes les adresses n'ont pas forcément d'octet
 - Ou n'ont pas les mêmes règles d'accès (lecture, écriture, exécution, etc.)
- À quoi correspond chaque adresse (RAM, ROM, périphérique, etc.)
- Autres considérations : boutisme et alignement (on y reviendra)

Segments et sections

- Zone continue en mémoire de code et données binaires
- Concept assemblage, édition de liens et chargement 🗹

GNU binutils

- Distinction linguistique
 - segment: pour l'édition de liens
 - section: pour le chargement
- objdump -h pour lister les segments d'un binaire

RARS

- Assembleur gère deux sections seulement
 - .text de 0x400000 à 0x10000000
 - .data de 0x10010000 à 0x10040000
- Ajoute des contraintes sur leur utilisation
- Configurable: Settings>Memory Configuration

Deux segments principaux

Text: Segment de code machine

- Directive .text
- Contient les instructions
- En lecture seule (en général)
 - ROM (système embarqué)
 - Zone mémoire protégée (système d'exploitation)

Data: Segment de données

- Directive .data
- Contient les directives de données
- En lecture-écriture (en général)
 - RAM
 - Zone mémoire non exécutable (système d'exploitation)

Text et Data RARS

0000 0000	
003F FFFF	
0040 0000	
	text
OFFF FFFF	
1000 0000	
1001 0000	
	data
1003 FFFF	
1004 0000	
FFFF FFFF	

Segments

Autres segments classiques

- .bss données non initialisées (pas en RARS)
- .rodata données en lecture seule (pas en RARS)
- Pile (stack) et tas (heap): plus tard

Pourquoi des segments

- Plus de liberté et optimisation dans l'organisation du programme en mémoire
- ROM vs. RAM
- Protections: lecture seule, pas d'exécution
- Partage entre processus
- Détails en INF3173

Directives de données

À mettre dans les sections .data

Données de tailles fixes

- .byte valeurs génère des octets
- .half valeurs génère des paquets de 2 octets
- .word valeurs génère des paquets de 4 octets
- .dword valeurs génère des paquets de 8 octets où valeurs est une ou plusieurs constantes entières

Utilisation

Variables globales et constantes

Exemple

- .data
- .byte 42, 100, 0, -10 # 2A 64 00 F6

Boutisme

Donnés de taille fixe en mémoire

- Donnée stockée en mémoire sur plusieurs octets contigus
- L'adresse d'une donnée et celle du premier octet
- Les autres octets suivent dans l'ordre

Exemple

Problème: quelle est la valeur en mémoire de chaque octet ?

Boutisme (endianness)

L'ordre de stockage en mémoire des octets d'un mot

Gros boutisme		Petit boutisme		
1000FFFF			1000FFFF	• • • •
10010000	DE	\leftarrow adresse \rightarrow	10010000	EF
10010001	AD		10010001	BE
10010002	BE		10010002	AD
10010003	EF		10010003	DE
10010004			10010004	•••

Boutisme des architectures

- x86 : petit-boutiste
- PEP/8 : gros-boutiste
- ARM et RISC-V: bi-boutiste (petit-boutiste par défaut)

Exemple

- .data .byte 42, 100, 0, -10 # 2A 64 00 F6 .half 42, 100, 0 , -10 # 2A 00 64 00 00 00 F6 FF
 - .byte: octets indépendants (pas de boutisme)
 - .half: paquets de deux octets, petit-boutiste

Alignement

Alignement en mémoire

• Les accès mémoire se font en groupes d'octets Meilleure performance et garanties de cohérence

Alignement naturel

L'adresse est un multiple de la taille de la donnée: tout se passe bien

- ullet 16 bits o adresse multiple de 2
- ullet 32 bits o adresse multiple de 4
- ullet 64 bits o adresse multiple de 8
- ullet 8 bits o toujours naturellement aligné

Désalignement

L'adresse n'est pas un multiple de la taille de la donnée

- Possiblement plus lent
 Plusieurs accès mémoire & redécoupage des octets
- Possiblement interdit Trop couteux et cohérence difficile (atomicité, caches)

Note: les compilateurs gèrent automatiquement l'alignement

Directive d'alignement

Directive .align n

- Force localement un alignement 2^n
- n=1 pour halfword, n=2 pour word, etc.
- Insère des 00 si nécessaire

```
.data
```

```
.byte 42  # 2A
.align 2  # 00 00 00
.word 100  # 64 00 00 00
.align 2  # deja aligné: ne fait rien
```

Alignement implicite

L'assembleur peut aligner naturellement les données fixes

- RARS le fait
- GNU assembler ne le fait pas

Lectures et écritures

Architectures load-store vs register-memory

Architecture load-store (ou register-register)

- Seules des instructions spécifiques accèdent à la mémoire
- Les autres instructions travaillent avec des registres
- Exemples: RISC-V et ARM

Architecture register-memory

- La plupart des instructions peuvent accéder à la mémoire
- Exemple: x86

Instructions de lecture (load), Type I

• 1b rd, offset(rs1) lit un octet (load byte)

Où

- rd registre de destination
- rs1 est le registre de base
- offset est le décalage (possiblement zéro)
 - 12 bits signé
- L'adresse mémoire effective est rs1+offset (base+décalage)
- Note: la notation parenthésée est étrange mais historique

Exemple

1b s0, 4(s1)

- s0 est le registre de destination
 - La valeur lu lui sera assigné
- s1 contient l'adresse de base du contenu à lire
- 4 est le décalage apposé à l'adresse qui était présente dans le registre s1
- Le contenu de s0 sera assigné avec le contenu de l'adresse s1+4
- Note: s0 va recevoir et contenir une valeur. s1 contient une adresse de la mémoire à aller lire

Gestion du signe

Charger moins de bits qu'en contient un registre

Exemple octet 0A ou FF dans un registre 64 bits

- Que faire des bits restants du registre ?
- Rappel: les bits n'ont pas de sémantique intrinsèque

Stratégies possibles

- Mettre à 0: chargement non signé
 - OA \rightarrow OOOOOOOA \rightarrow 10
 - FF \rightarrow 000000FF \rightarrow 255
- Copier le bit de poids fort: extension de signe
 - OA \rightarrow OOOOOOOA \rightarrow 10
 - ullet FF ightarrow FFFFFFF ightarrow -1
- Laisser les bits tel quels
 - Pas disponible en RISC-V

Instructions de lecture (load), Type I

Versions signées (extension de signe)

- 1b rd, offset(rs1) lit un octet (byte)
- 1h rd, offset(rs1) lit un demi mot (2 octets, halfword)
- lw rd, offset(rs1) lit un mot (4 octets, load word)
- 1d rd, offset(rs1) lit un double mot (8 octets, RV64)

Versions non signées

- lbu rd, offset(rs1) lit un octet (byte unsigned)
- lhu rd, offset(rs1) lit un demi mot
- lwu rd, offset(rs1) lit un mot (RV64)

Question

- Pourquoi pas de 1du en RV64 ?
- Pourquoi 1wu seulement en RV64 ?

Instruction d'écriture (store), Type S

- sb rs2, offset(rs1) écrit un octet (byte)
- sh rs2, offset(rs1) écrit un demi mot (2 octets, halfword)
- sw rs2, offset(rs1) écrit un mot (4 octets, store word)
- sd rs2, offset(rs1) écrit un double mot (8 octets, RV64)

Оù

- rs2 registre source
- rs1 est le registre de base
- offset est le décalage (possiblement zéro)
 - 12 bits signé
- L'adresse mémoire effective est rs1+offset (base+décalage)
- Note: attention la destination mémoire est à droite
- Question: y a-t-il une version non-signée?

Tailles d'architectures



RV32 vs. RV64

- La taille des registres dépend de l'architecture
- Mais pour lire ou sauvegarder **entièrement** un registre, il faut:
 - Avoir préparé la bon nombre d'octets en mémoire
 - Utiliser les bonnes instructions lw/sw (RV32) ou ld/sd (RV64)

Question

Que se passe-t-il si

- On utilise lw/sw en RV64 ?
- On utilise 1d/sd en RV32 ?

Boutisme et alignement

Boutisme

- Lectures et écritures utilisent le boutisme de l'architecture
- Petit-boutiste par défaut en RISC-V

Alignement

En RISC-V, les lectures et écritures non naturellement alignés sont

- Soit interdites
- Soit fortement découragées (exécution plus lente, simulation logicielle)

Autres contraintes d'environnement 🚱



- Certaines adresses peuvent ne pas exister
- Où avoir des droits limités (pas écrivable, pas exécutable, etc.)
- Les détails une autre fois

Exercice (sans pseudoinstructions)

.data

```
.word 40, 100, 0
```

- Faire la somme des deux premiers mots à l'adresse 0x10010000
- Écrire la somme sur le 3e mot

Exercice (sans pseudoinstructions)

.data

```
.word 40, 100, 0
```

- Faire la somme des deux premiers mots à l'adresse 0x10010000
- Écrire la somme sur le 3e mot

.text

```
li a0, 0x10010000 # data

lw a1, 0(a0) # lit premier mot

lw a2, 4(a0) # lit second mot

add a1, a1, a2 # somme

sw a1, 8(a0) # écrit 3e mot
```

Pseudoinstructions

Base ou décalage optionnel

- ullet lb rd, offset ightarrow lb rd, offset(x0)
 - offset sur 12 bits
- ullet lb rd, (rs1) ightarrow lb rd, 0(rs1)
- Fonctionne aussi pour tous *load* et *store*

Adresse absolue (RARS seulement)

Q

- 1b rd, adresse
 - lui rd, adresse[31:12]; lb rd, adresse[11:0](rd)
 - Fonctionne aussi pour tous les load
- sb rs2, adresse, rt
 - nécessite un registre temporaire
 - lui rt, adresse[31:12]; sb rs2, adresse[11:0](rt)
 - Fonctionne aussi pour tous les store
- Note: RARS seulement (non géré par GNU assembler)

Mauvaises pratiques

Mettre des adresses explicites (en dur) c'est mal

- Dépendent de l'environnement d'exécution
- Compliquée à calculer à la main
- Compliquée à mettre à jour à la main
- Rôle de la constante non intuitive
- Défaut de code: nombre magique (magic number)

Solution

- Nommer une adresses mémoire avec une étiquette
- L'assembleur (ou l'éditeur de liens) déterminera sa valeur

Pseudoinstructions à étiquettes

- lb rd, label
- sb rs1, label, rt

Οù

- rt: un registre temporaire utilisé
- label: une étiquette définie quelque part (même après)
- Fonctionne aussi pour tous *load* et *store*
- Question: pourquoi 1b n'a pas besoin de registre temporaire ?

Exemple

```
lw a1, taille
  addi a1, a1, 1
  sw a1, taille, t0
.data
taille: .word 42
```

Exercice (avec étiquettes)

.data

a: .word 40 b: .word 100 c: .word 0

• Écrire à l'adresse c la somme des mots aux adresses a et b

Exercice (avec étiquettes)

```
.data
a: .word 40
b: .word 100
c: .word 0
```

• Écrire à l'adresse c la somme des mots aux adresses a et b

```
lw a1, a # lit premier mot
lw a2, b # lit second mot
add a1, a1, a2 # somme
sw a1, c, t0 # écrit 3e mot
```

Pseudoinstructions avec étiquettes (la vérité)



Plusieurs formes possibles, en fonction de:

- Les outils: assembleur, éditeur de liens, chargeur dynamique
- La configuration et la version de ces outils
- L'environnement d'exécution ciblé
- Les détails une autre fois...

RARS

Combine avec auipc

```
lw rd, label # devient →
auipc rd, H
lw rd, L(rd)
```

- H et L codent la différence entre l'étiquette et l'instruction
- L'assembleur sait déterminer cette différence d'adresse
- Fonctionne aussi avec les autres formes de load et store

Pseudoinstruction de calcul d'adresse

load address, pseudoinstruction

- la rd, label
- Range dans rd l'adresse de label
- Vraies instructions dépendantes des outils
- Mal nommée?
 - Calcule l'adresse effective d'une étiquette
 - L'addresse effective dépend si le code est indépendant de sa position (PIC / Position Independent Code)
 - Mais ne fait pas de lecture en mémoire à cette adresse

RARS

- Utilise aussi auipc
- auipc rd, H addi rd, rd, L
- Les valeurs de H et L sont calculés automatiquement par l'assembleur

Exercice (avec la)

```
.data mots: .word 40, 100, 0
```

- Faire la somme des deux premiers mots de l'adresse mots
- Écrire la somme sur le 3e mot

Exercice (avec la)

.data mots: .word 40, 100, 0

- Faire la somme des deux premiers mots de l'adresse mots
- Écrire la somme sur le 3e mot

.text

```
la a0, mots # Adresse de base
lw a1, 0(a0) # lit premier mot
lw a2, 4(a0) # lit second mot
add a1, a1, a2 # somme
sw a1, 8(a0) # écrit 3e mot
```

Équivalence

```
lb s0, label
est équivalent à
la s1, label
lb s0, (s1)
```

Ce qui est plus performant si on réutilise le registre pour faire plusieurs accès mémoire

Modèle mémoire



Mémoire cache de processeur

- Mémoire très rapide, duplique (très) partiellement la mémoire
- Les processeurs optimisent ainsi les accès mémoire
- Retards et/ou réordonnancement des accès effectifs
- Les détails en INF4170 Architecture des ordinateurs

Architectures parallèles

- Présence de multiprocesseurs et/ou multicœurs
- Plusieurs programme s'exécutent en même temps
- Partagent mémoire et périphériques
- Mais pas forcément les caches
- Chaque cœur a potentiellement sa propre vision de la mémoire
- Les détails en INF5171 Programmation concurrente et parallèle

Synchronisation des accès mémoire



Barrière mémoire (memory barrier, memory fence)

Impose l'ordre sur les accès effectifs

- Les accès effectifs qui précèdent la barrière
- sont garanties réalisés avant
- les accès effectifs qui suivent la barrière

Autrement, rien n'est vraiment garanti (pour plus de performance)

Instruction de barrière mémoire RISC-V



fence pred, succ (type spécial)

- pred et succ sont une combinaison de iorw
 - i: entrée périphérique (input)
 - o: sortie périphérique (output)
 - r: lecture mémoire (read)
 - w: écriture mémoire (write)
- Impose aux accès pred d'avoir lieu avant les accès succ
- Note: pas vraiment supporté par RARS

Pseudoinstruction fence tout seul

- Équivalent à fence iorw, iorw
- Barrière totale mémoire et entrées-sorties

Tableaux et pointeurs

Pointeurs en général



- **Pointeur** = donnée qui contient une **adresse** mémoire D'une donnée ou d'une instruction par exemple
- Rappel: utiliser l'hexadécimal pour représenter une adresse

Manipulation directe de la mémoire

- Utilisation restreinte dans de nombreux langages de haut niveau
- Problèmes de robustesse et de cohérence

Les pointeurs sont des données

- On peut les modifier, les stocker, les transmettre
- On peut les calculer (arithmétique)

Pointeurs en assembleur: trivial

- On manipule déjà directement des adresses
- Opérande des instructions load et store
- Opérande des instructions de branchement (b*, j*)

Pointeurs en mémoire

- Adresse mémoire stockée en mémoire
- Taille d'un pointeur dépend de l'architecture
 - 4 octets (RV32) ou 8 octets (RV64)

```
.data
ptr:    .dword valeur
valeur:    .dword 42
.text
    la s0, ptr # s0 = ptr
    ld s1, (s0) # s1 = valeur
    ld s2, (s1) # s2 = 42
```

Tailles d'architectures



RV32 vs. RV64

- La taille des pointeurs dépend de l'architecture
- Donc pour stocker ou lire un pointeur en mémoire, il faut:
 - Avoir préparé le bon nombre d'octets en mémoire
 - Utiliser les bonnes instructions lw/sw (RV32) ou ld/sd (RV64)

Tableaux



Tableaux en assembleur

- Une séquence de valeurs en mémoire
- L'adresse du tableau est l'adresse de la première valeur

Le programme assembleur gère tous les détails

- Pas d'instruction machine ou assembleur dédiée
- Données
 - La longueur du tableau
 - La taille des éléments du tableau (octets, mots, etc.)
 - L'alignement des éléments du tableau
- Code
 - Le calcul des adresse des éléments
 - L'accès aux éléments
 - Les tests des bornes du tableau

Données de tableaux

- Directives de données de taille fixe (.word, etc.)
- Directive .eqv pour la taille (en octets et/ou en nombre d'éléments)

```
.data
```

```
# Tableau t1 de 5 octets
t1: .byte 10, 0, 30, -8, 2
    .eqv t1len, 5 # taille du tableau t1
```

```
# Tableau t2 de 5 mots: 20 octets en tout
.align 2 # pour GNU assembler afin d'aligner t2
```

```
t2: .word 10, 0, 30, -8, 2
.eqv t2len, 20 # taille du tableau t2 en octets
```

Stratégies alternatives

- Étiquette de fin de tableau
- Sentinelle de fin de tableau

Exercices

Écrire un programme somme.s

• Calculer et affiche la somme des 10 éléments du tableau .data

```
tableau: .word 10, 10, -6, 20, 1, 1, 8, 800, -800, -2
```

Écrire un programme maximum.s (en lab)

 Modifier le programme pour afficher également l'élément le plus grand du tableau

Chaînes de caractères

Code ASCII, 1968

Rappel

- Chaque caractère représenté par un code de 7 bits
- ASCII = American Standard Code for Information Interchange

Chaîne ASCII

- Une chaîne est une séquence de caractères (en mémoire, sur disque, etc.)
- Convention C: le caractère nul ('\0') marque la fin d'une chaîne
- Autre stratégie: taille connue et maintenue quelque part

B: B:						° ° °	٥ ،	0,0	۰,	00	0,	۱ _۱	1 1
	b₄	b₃ ↓	b₂	₽.	Column	0	- 1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	Р	`	р
	0	0	0	1	1	SOH	DCI	!	- 1	Α	Q	а	q
	0	0	_	0	2	STX	DC2	"	2	В	R	b	r
	0	0	_	1	3	ETX	DC3	#	3	С	S	С	s
	0	-	0	0	4	EOT	DC4	\$	4	D	Т	d	t
	0	1	0	1	5	ENQ	NAK	%	5	Ε	υ	е	u
	0	1	_	0	6	ACK	SYN	8.	6	F	V	f	v
	0	1	-	ı	7	BEL	ETB	,	7	G	W	g	w
	ı	0	0	0	8	BS	CAN	(8	Н	×	h	x
	ı	0	0	ı	9	нт	EM)	9	I	Y	i	У
	1	0	_	0	10	LF	SUB	*	:	J	Z	j	z
	1	0	1	ı	П	VT	ESC	+	;	K	[k	{
	1	ı	0	0	12	FF	FS	,	<	L	١	1	
	ı	1	0	ı	13	CR	GS	_	=	М]	m	}
	ı	1	1	0	14	so	RS		>	N	^	n	~
	ı	ı	ı	1	15	SI	US	/	?	0	_	0	DEL

Directive de donnés variables

Chaines littérales

- ascii chaine
 Alloue un octet pour chaque caractère de la chaine
- .asciz chaine
 Même chose + un octet à 0 à la fin: caractère nul ('\0')
- .string chaine Synonyme de .asciz

```
.ascii "Hello" # 48 65 6C 6C 6F
.asciz "Hello" # 48 65 6C 6C 6F 00
.string "Hello" # 48 65 6C 6C 6F 00
```

Boutisme et alignement

• Pas de contraintes car c'est des séquences d'octets

Directive de donnés variables

Réservation d'espace

- .space nombre
- Alloue nombre octets à 0
- Utilisation: données non initialisées, tableau non initialisé et réservation de tampons
- On préfère utiliser le segment dédié .bss
 - .bss segment de données non initialisées
 - Mais .bss n'existe pas en RARS :(

.space 5 # 00 00 00 00 00

Entrées-sorties (niveau 2)

Appels système RARS

- PrintString: a7: 4
 - Entrée a0: adresse de la chaine à afficher
 - S'arrête au premier octet nul '\0' rencontré
- ReadString: a7: 8
 - Entrée: a0 adresse d'un espace mémoire (tampon, buffer)
 - Entrée: a1 taille de l'espace mémoire
 - Sortie: aucune, mais rempli le tampon avec une ligne lue Insère également un \0 à la fin

Routines de libs.s

- printString
- readString
- Sémantiques suffisamment équivalentes :)

Exemple: helloworld.s

```
.text
    li a7, 4 # PrintString
    la a0, helloStr
    ecall
    li a7, 10 # Exit
    ecall
.data
helloStr: .string "Hello, World!\n"
```

Exercices

- Recoder helloworld2.s avec la routine printString (de libs.s)
- Coder echo.s qui lit une ligne et l'affiche deux fois (RARS)
- Recoder echo2.s avec les routines de libs.s

Entrées-sorties (niveau 2)

Appels système RARS et Linux

- write: a7: 64
 - a0 ← descripteur de fichier (mettez 1)
 - a1 ← adresse mémoire du tampon
 - a2 ← taille à écrire
 - Sortie: a0: taille effectivement écrite ou -1 si erreur
- read: a7: 63
 - a0 ← descripteur de fichier (mettez 0)
 - a1 ← adresse mémoire du tampon
 - a2 ← taille maximum à lire
 - Sortie: a0: taille effectivement lue ou -1 si erreur

Exercices

- Recoder helloworld3.s avec l'appel système Unix write
- Recoder echo3.s avec les appels systèmes Unix

Conclusion

Organisation de la mémoire

Segments

• Directives: .text et .data

Données globales statiques

 directives .byte, .half, .word, .dword, .align, .ascii, .asciz, .string, .space

Lectures et écritures en mémoire

- Instructions: 1b, 1bu, 1h, 1hu, 1w, 1wu, 1d, sb, sh, sw, sd
- Pseudoinstruction: la

Attention

- Boutisme
- Alignement

Mémoire avancée



Allocations dynamiques

Pile et tas: plus tard

Pagination

- Mécanisme évolué utilisés par les systèmes d'exploitation
- Encore plus tard
 - INF3173 Principes de systèmes d'exploitation
 - INF4170 Architecture des ordinateurs
- Mode privilégié (volume 2) en RISC-V

Opérations atomiques

- INF3173 Principes de systèmes d'exploitation
- INF5171 Programmation concurrente et parallèle
- Extension "A" en RISC-V

La prochaine fois

Calculs

- Plus de boucles
- Plus d'arithmétique
- Plus de programmes